TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection

Tielei Wang[1,2], Tao Wei[1,2], Guofei Gu[3], Wei Zou[1,2*]
[1]Key Laboratory of Network and Software Security Assurance (Peking University),
Ministry of Education, Beijing 100871, China
[2]Institute of Computer Science and Technology, Peking University
[3] Department of Computer Science & Engineering, Texas A&M University
{wangtielei, weitao, zouwei}@icst.pku.edu.cn, guofei@cse.tamu.edu

***Abstract*—Fuzz testing has proven successful in finding security vulnerabilities in large programs. However, traditional fuzz testing tools have a well-known common drawback: they are ineffective if most generated malformed inputs are rejected in the early stage of program running, especially when target programs employ checksum mechanisms to verify the integrity of inputs. In this paper, we present TaintScope, an automatic fuzzing system using dynamic taint analysis and symbolic execution techniques, to tackle the above problem. TaintScope has several novel contributions: 1) TaintScope is the first checksum-aware fuzzing tool to the best of our knowledge. It can identify checksum fields in input instances, accurately locate checksum-based integrity checks by using branch profiling techniques, and bypass such checks via control flow alteration. 2) TaintScope is a *directed* fuzzing tool working at X86 binary level (on both Linux and Window). Based on fine-grained dynamic taint tracing, TaintScope identifies which bytes in a well-formed input are used in security-sensitive operations (e.g., invoking system/library calls) and then focuses on modifying such bytes. Thus, generated inputs are more likely to trigger potential vulnerabilities. 3) TaintScope is fully automatic, from detecting checksum, directed fuzzing, to repairing crashed samples. It can fix checksum values in generated inputs using combined concrete and symbolic execution techniques.**

We evaluate TaintScope on a number of large real-world applications. Experimental results show that TaintScope can accurately locate the checksum checks in programs and dramatically improve the effectiveness of fuzz testing. TaintScope has already found 27 previously unknown vulnerabilities in several widely used applications, including Adobe Acrobat, Google Picasa, Microsoft Paint, and ImageMagick. Most of these severe vulnerabilities have been confirmed by Secunia and oCERT, and assigned CVE identifiers (such as CVE-2009-1882, CVE-2009-2688). Corresponding patches from vendors are released or in progress based on our reports.

***Keywords*-fuzzing; dynamic taint analysis; symbolic execution;**

I. Introduction

As a well-known software testing technique, fuzz testing or fuzzing [51] has proven successful in finding bugs and security vulnerabilities in large software. The idea behind fuzzing is very simple: generating malformed inputs and feeding them to an application; if the application crashes or hangs, a potential bug/vulnerability is detected. A number of severe software vulnerabilities have been revealed by fuzzing techniques [64], [58]. For example, with the help of fuzzing tools, “Month of Browser Bugs” [10] and “Month of Kernel Bugs” [11] published bugs in various browsers and kernels on a daily basis for the month of July and November in 2006.

Since exhaustive enumeration of an application’s input space is typically infeasible, there are two main approaches to obtain malformed inputs: *data mutation* and *data generation* [58]. Mutation-based fuzzing tools generate test cases by randomly modifying well-formed inputs. However, most malformed inputs from such blind modifications will be dropped at an early stage of program running if the target program employs a ***checksum*** mechanism to verify the integrity of inputs. The effectiveness of these fuzzing tools is heavily limited by checksum-based integrity checks.

Recently, symbolic execution and constraint solving based whitebox fuzzing systems (such as KLEE [27], SAGE [43], SmartFuzz [52], EXE [28], CUTE [62], DART [42]) can substitute all program inputs with symbolic values, gather input constraints on a program trace and generate new inputs that can drive program executions along different traces. These systems are able to provide good code coverage and have proven to highly improve the effectiveness of traditional fuzzing tools. However, current symbolic execution engines and constraint solvers still cannot accurately generate and solve the constraints that describe the complete process of complex checksum algorithms [63]. In a word, such whitebox fuzzing systems also cannot automatically generate inputs which satisfy the checksum-based integrity constraints.

For generation-based fuzzing tools (such as SPIKE [18], Peach [15], PROTOS [60]) which construct malformed input data from predefined format specifications, the cost of generating production rules used by fuzzing tools is expensive, especially when the format specifications are undocumented and the source code of the application is not available. Recently, several protocol reverse engineering techniques [26], [47], [34], [31] are proposed to automatically extract

*Corresponding author

1081-6011/10 $26.00 © 2010 IEEE
DOI 10.1109/SP.2010.37

input format specification (even protocol state machine) and translate them into fuzzing specifications. However, they are not able to reverse engineer the checksum algorithms. Thus, the constructed input according to such reverse-engineered protocol specification still can be rejected by integrity checks. In addition, none of such systems explicitly discuss how to bypass checksum checks for fuzzing.

In this paper, we present TaintScope, a checksum-aware directed fuzzing system based on dynamic taint analysis and symbolic execution. The key idea behind TaintScope is that the taint propagation information during program execution can be used to detect and bypass checksum-based integrity checks, and to direct malformed test cases generation. TaintScope can further fix checksum fields in malformed test cases by using combined concrete and symbolic execution techniques. More specifically, this paper makes the following contributions.

First, we propose a novel approach to infer whether/where a program checks the integrity of input instances and can perform *checksum-aware fuzzing*. The high level intuition of the approach is illustrated in Figure 1. Checksum-based integrity checks behave like a classifier: while all well-formed inputs pass the integrity checks, most malformed inputs fail to pass because of integrity violations. Thus, we assume that there are special branch predicates in the program, corresponding to integrity checks: when the program runs with well-formed inputs, these branch predicates are always True/False; however, when the program runs with malformed inputs, these branch predicates are always False/True. By profiling program traces, TaintScope builds four predicate sets $\mathcal{P}_1$, $\mathcal{P}_0$, $\mathcal{P}'_1$ and $\mathcal{P}'_0$, where $\mathcal{P}_1/\mathcal{P}_0$ contain the always-true/always-false predicates when the program runs with well-formed inputs, and $\mathcal{P}'_1/\mathcal{P}'_0$ contain the always-true/always-false predicates when the program runs with malformed inputs, respectively. The predicates in $(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)$ usually correspond to checksum checks.

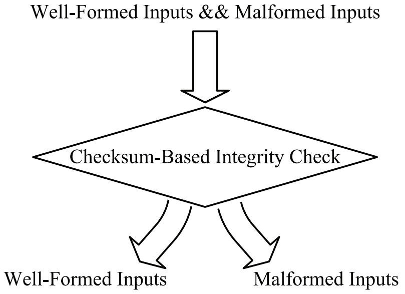
Well-Formed Inputs && Malformed Inputs
Checksum-Based Integrity Check
Well-Formed Inputs
Malformed Inputs

Figure 1. The intuition of locating checksum check points

Unlike existing fuzzing systems, TaintScope can enforce the alteration of the target program's execution trace at located integrity check points, as if the generated test cases did not violate integrity checks. We call this *checksum-aware fuzzing*. Checksum-aware fuzzing can prevent generated test cases from being prematurely dropped, and this feature is helpful to trigger deep subtle errors in the rest of the program.

Second, TaintScope is a *directed* fuzzing tool with fine-grained taint analysis at byte level. In contrast to traditional taint analysis with 0/1 taint labels [57], TaintScope marks each input byte with a unique label (i.e., the byte's position in an input instance), monitors how the target application accesses and uses the input data, and tracks the propagation of these labels throughout the execution of the program. Consequently, TaintScope is able to accurately identify which input bytes can flow into security-sensitive points (e.g., memory allocation function `malloc()`, string manipulation function `strcpy()`). Thus, given a well-formed input instance, TaintScope does not blindly modify the whole input, but focuses on modifying the set of bytes that could affect the values used in security-sensitive points (e.g., system/library calls). Directed fuzzing has three advantages: 1) it dramatically reduces the cardinality of the mutation space because we specially focus on modifying a small part of the original input instance; 2) minor modification conducted by directed fuzzing usually does not break the syntactic structure in the original input instance; 3) the malformed inputs generated by directed fuzzing are more likely to expose security vulnerabilities. Our directed fuzzing idea is motivated by BuzzFuzz [40]. However, quite different from BuzzFuzz that requires access to an application's source code, TaintScope can directly work on both Linux and Windows binary executables.

Third, TaintScope is fully automatic, from detecting checksum, directed fuzzing, to repairing crashed samples. TaintScope can fix checksum fields in generated test cases using combined concrete and symbolic execution techniques. Instead of treating all input bytes as symbolic values, TaintScope only substitutes the checksum fields in test cases with symbolic values (i.e., leave the majority input bytes as concrete values), and collects trace constraints on checksum fields. Original complex trace constraints are simplified to simple constraints. By solving such simple constraints, TaintScope can repair generated test cases.

Finally, we evaluate TaintScope on a number of large real-world applications. Experimental results show that TaintScope can accurately locate the checksum checks in programs and dramatically improve the effectiveness of fuzz testing. TaintScope has already found 27 previously unknown vulnerabilities in several widely used applications, including Adobe Acrobat, Google Picasa, and ImageMagick. We have reported our finding and contacted corresponding vendors. Most of these vulnerabilities have been confirmed by Secunia and oCERT, and assigned CVE identifiers (such as CVE-2009-1882, CVE-2009-2688). Corresponding patches from vendors are released or in preparation based on our reports.

II. Overview

A. Problem Scope and Terminologies

Checksums [2] are a common way to check the integrity of data and are widely used in network protocols (e.g., TCP/IP) and many file formats (e.g., ZLIB [35], PNG [23]). There are many sophisticated checksum algorithms used in practice, such as Adler-32, CRC (cyclic redundancy checks) series, and MD5. In this paper, we focus on checksums that are designed to protect against mainly *accidental* errors that may have been introduced during transmission or storage, instead of those designed to protect against *intentional* data alteration such as keyed cryptographic hash algorithms. We leave the later as our future work.

In general, the basic pattern to check the integrity of an input instance is to recompute a new checksum of the input instance and compare it with the checksum attached in the input instance. A mismatch indicates a corrupted input instance. For easy of representation, we use C_r and D to represent raw data in the checksum field and other input data that are protected by the checksum field in an input instance, respectively.

Without loss of generality, we assume that the checksum check condition is equivalent to the condition P: $Checksum(D) == T(C_r)$, where $Checksum()$ denotes the complete process of checksum algorithms and $T()$ denotes the transformation function that is used to transform C_r before integrity checks. For instance, the attached checksums are stored as octal numbers in the Tar format [19], or stored as hexadecimal numbers in the Intel HEX format [7]. These raw data C_r need to be converted into proper forms before being used in integrity checks. $T()$ is used to describe the transformation process.

We assume that there are special branch predicates in the program, corresponding to integrity checks P. The predicate P is always true/false when inputting well-formed instances, whereas the predicate is always false/true when inputting malformed instances. One of our goals is to accurately locate potential integrity check points in a binary program rather than identify the checksum algorithm themselves.

Furthermore, TaintScope specially focuses on modifying the input bytes that can affect the arguments of important API functions, such as memory management functions, string manipulation functions. In this paper, we refer to such input bytes as "*hot bytes*". Our another goal is to automatically identify hot bytes in a well-formed input.

B. Motivating Example and System Overview

As a motivating example, Figure 2 shows an example input format. This example input format presents a common image format: `MagicNum` field declares the file format; `FileSize` field indicates the real size of an input image; `Width` and `Height` fields mean the width and the height of an input image. In this format, image data are stored in row-major order. Note that the format ends with a checksum field. A four-byte checksum is calculated on the preceding image content in the file for integrity protection.

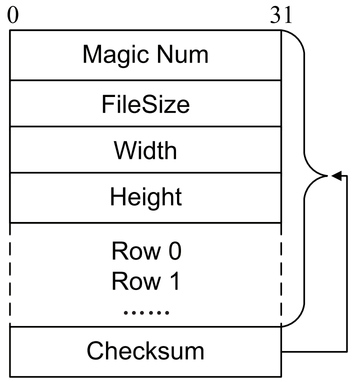

Figure 2. Example input format

```
 1  void decode_image(FILE* fd){
 2   ...
 3   int length            = get_length(fd);
 4   int recomputed_chksum = checksum(fd, length);
 5   int chksum_in_file    = get_checksum(fd);
     //line 6 is used to check the integrity of inputs
 6   if(chksum_in_file != recomputed_chksum)
 7     error();
 8   int Width    = get_width(input_file);
 9   int Height   = get_height(input_file);
10   int size = Width*Height*sizeof(int); //integer overflow
11   int* p   = malloc(size);
12   ...
13   for(i=0; i<Height;i++){// read ith row to p
14     read_row(p + Width*i, i, fd); //heap overflow
```

Figure 3. Example Code

Figure 3 shows an example code that parses inputs from the example format. The code first recomputes a checksum of an input file (line 4), reads the checksum stored in the file (line 5), and then compares the two values (line 6). If the two values mismatch, the code raises an error and exits immediately. Next, the code reads the `Width` and `Height` fields, then allocates memory of size "`Width*Height*sizeof(int)`". Finally, the code reads image data row by row into the allocated buffer. However, a specially crafted image containing large width and height values can cause an integer overflow in the above expression (line 10) and further lead to an insufficient memory allocation at line 11. A heap overflow will eventually occur when the code reads image data into memory, leading to a potential attack.

Traditional fuzzing methods such as randomly modifying a well-formed input file are unlikely to find such integer overflow vulnerability in Figure 3. Because a simple modification breaks the integrity of the original input, the failed integrity check at line 6 will cause most generated test cases to be rejected. Furthermore, even if there were no integrity

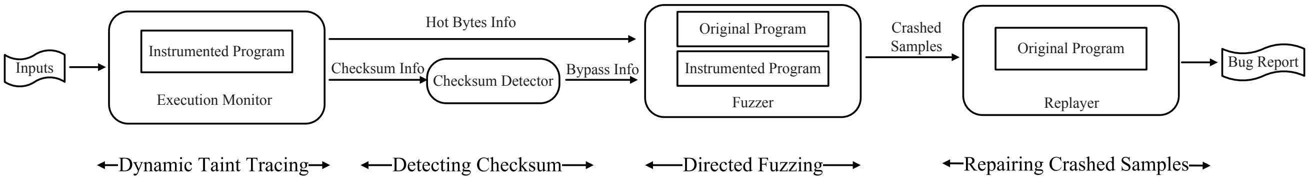

Figure 4. TaintScope System Overview

check at line 6, traditional fuzzing methods would also be too inefficient. In this example, only eight bytes (`Width` and `Height` fields) in an input file are key factors to trigger the integer overflow vulnerability; however, blind modifications need to explore the whole input space. The probability of exactly modifying `Width` and `Height` fields is very low.

In contrast, TaintScope leverages taint propagation information during program execution to detect checksum checks and generate test cases, which makes triggering vulnerabilities more likely. At a high level, TaintScope has four phases: *dynamic taint tracing*, *detecting checksum check points*, *directed fuzzing*, and *repairing crashed samples*, as shown in Figure 4. We use the program in Figure 3 to illustrate the basic workflow of TaintScope. Note that TaintScope neither depends on the source code nor on the input format specification.

Dynamic Taint Tracing. Given a binary program *P* to test, TaintScope first runs *P* with a well-formed input *I*. TaintScope uses an *execution monitor* to dynamically instrument the program *P* and monitor how the program *P* processes the input data *I*. In particular, the execution monitor records the following information: 1) which input bytes pollute the arguments of specified API functions; and 2) which input bytes each conditional jump instruction (e.g., `JZ`, `JE`, `JB`) depends on. The former is hot bytes information and the latter is checksum information. In addition to the arguments of specified API functions, the execution monitor can also be configured to monitor which input bytes can influence the execution context of any program point.

In this example, assume that the size of the input file *I* is 1024 bytes and analysts are interested in the size argument of memory allocation function `malloc`, the hot bytes report will include entries like this:

```
0x8048d5b: invoking malloc: [0x8,0xf]
```

which means the instruction at `0x8048d5b` calls function `malloc` and the size argument depends on input bytes in the range from `0x8` to `0xf`. Similarly, assume that branch statement line 6 in Figure 3 is compiled into a `JZ` instruction at address `0x8048d4f`, the checksum information report will include entries like this:

```
0x8048d4f: JZ: 1024: [0x0, 0x3ff]
```

which means the conditional jump instruction `JZ` at `0x8048d4f` depends on 1024 input bytes in the range from `0x0` to `0x3ff`.

Detecting Checksum Check Points. In this phase, TaintScope uses a checksum detector to locate potential checksum check points in the program *P*. We present the details of checksum detector in Section III-C. In this example, the checksum detector identifies the conditional jump instruction `JZ` at `0x8048d4f` as an integrity check. Meanwhile, the checksum detector generates a bypass rule, "`0x8048d5b: JZ: always-taken`", which means the branch instruction `JZ` at `0x8048d5b` needs to be always taken. Similar to Tupni [34], TaintScope can also identify checksum fields in each input instance.

Directed Fuzzing. In the third phase, a fuzzer module is responsible for generating malformed test cases and feeding them back to the target program. If the checksum detector does not generate bypass rules, the fuzzer directly feeds malformed test cases to the original program; otherwise, the fuzzer feeds malformed test cases to an instrumented program; according to the bypass rules in *Bypass Info*, the fuzzer alters the execution traces at checksum check points. In particular, all malformed test cases are constructed based on the hot bytes information. The output of this phase is the test cases that could cause the program to crash or consume 100% CPU.

In this example, the hot byte information directs the fuzzer to modify input bytes in the range from `0x8` to `0xf`. Before executing the conditional jump instruction `JZ` at `0x8048d4f`, the fuzzer sets the condition code flag `ZF` in the `eflags` register to an opposite Boolean value. Thus, all generated test cases also pass the checksum check and are more likely to trigger the integer overflow vulnerability in the program.

Repairing Crashed Samples. For the test cases that cause the instrumented program to crash, TaintScope needs to fix checksum fields in the test cases. Note we do not apply checksum fixing in the previous phase for every fuzzing test case because the fixing is relatively expensive (time-consuming). It makes more sense to perform a delayed repair on only a small number of malformed test cases that actually cause the program to crash/hang. Given a malformed test case, TaintScope only treats the checksum value bytes as symbolic values (i.e., leave the majority input bytes as concrete values) and collects trace constraints. After the execution of checksum check point, TaintScope tries to generate a new test case that can execute a different branch. In this case, let the original complex trace condition be $Checksum(D) == T(C_r)$. If both the raw data in checksum fields C_r and other data D are symbolic values, for complex checksum algorithms such as MD5, this constraint cannot be solved. However, in our scenario, only C_r corresponds to symbolic values and $Checksum(D)$ is a runtime determinable constant value. Thus, the complex constraint can be simplified to a simple one, which can be solved with current solvers such as STP [39]. Finally, if the new test case can still cause the original program to crash, a potential vulnerability is detected. Note that this phase can be ignored if the fuzzer directly tests the original program (in the previous phase).

III. System Design and Implementation

In this section, we describe the detailed design and implementation of TaintScope. First, we introduce our fine-grained dynamic taint tracing technique in Section III-A. Next, we present the approaches to identify hot bytes and checksum check points in Section III-B and Section III-C, respectively. Then, we discuss our checksum-aware directed fuzzing technique in Section III-D. We introduce the combined concrete and symbolic execution technique for repairing crashed samples in Section III-E. Finally, we introduce the implementation of TaintScope system in Section III-F.

A. Fine-grained Dynamic Taint Tracing

The execution monitor module implements a fine-grained dynamic taint analysis at byte level. Traditional taint analysis systems such as [57] [61] specially focus on the propagation of all untrusted data and thus mark all input data with a single taint label. In contrast, the execution monitor assigns each input byte a unique label and tracks the propagation of these labels throughout the execution of the program. The execution monitor is built on top of PIN [49], a tool for the dynamic instrumentation of programs. The execution monitor supports both Linux and Windows binary executables.

Taint Source. Analysts can specify a filename or an IP address as a taint source. All data read from the file or received from the IP address are marked. To do this, the execution monitor makes the use of `PIN_AddSyscallEntryFunction` and `PIN_AddSyscallExitFunction` PIN APIs to intercept relevant system calls. For example, for file tainting on Linux systems, the execution monitor intercepts system calls such as `open`, `mmap`, `read`, `lseek`, and `close`. When the specified file is successfully opened, the execution monitor records the returned file descriptor `fd`. Each time when the system reads from file descriptor `fd`, the execution monitor scans the input buffer, and assigns each byte in the input buffer with its offset in the file. According to the return value of `read` and `lseek`, the execution monitor updates the file offset. In addition, when the file descriptor `fd` is closed with `close`, the execution monitor does not track it again.

Taint Propagation. Typically, there are two kinds of dependence relationships to consider: data-flow and control-flow dependencies. For data-flow dependencies, the execution monitor instruments data movement instructions (e.g., `MOV`, `PUSH`), arithmetic instructions (e.g., `SUB`, `ADD`), and logic instructions (e.g., `AND`, `XOR`). The execution monitor taints all values written by an instruction with the union of all taint labels associated with values used by that instruction. Note that the `eflags` register is also considered. For example, assume that the taint labels associated with `eax` and `ebx` are {0x6, 0x7} and {0x8, 0x9}, respectively, after the execution of instruction "`ADD eax, ebx`", the taint labels on `eax` would be {0x6, 0x7, 0x8, 0x9}; since instruction `ADD` also affects the `eflags` register, the taint labels on the `eflags` are updated to {0x6, 0x7, 0x8, 0x9}. For another example, assume the taint label associated with `ecx` is {0x100} and `0x8000000` is the base address of an untainted array, after the execution of this instruction "`MOV eax, [0x8000000+ecx*4]`", the `eax` would be associated with {0x100}. Our current execution monitor does not consider control-flow dependencies. We leave that as our future work.

B. Identification of Hot Bytes

Hot bytes refer to the input bytes that affect the values used in security-sensitive operations. Based on the fine-grained dynamic taint analysis described in Section III-A, the execution monitor can further identify hot bytes in a well-formed input. By default, the execution monitor checks which input bytes can pollute the arguments of memory allocation functions (e.g., `malloc`, `realloc`) and string functions (e.g., `strcpy`, `strcat`). Analysts can configure the execution monitor to check other functions, too.

The execution monitor utilizes PIN's routine instrumentation capability to hook target functions. Before target functions are executed, the execution monitor checks whether their arguments are tainted. The taint marks associated with the arguments clearly reveal which input bytes can affect such arguments. The execution monitor logs such hot bytes information to direct malformed input generation.

C. Locating Checksum Check Points and Checksum Fields

Checksums, a simple error-detection scheme, are a main challenge that traditional fuzzing tools cannot overcome. In general, the integrity of input data can be checked by recomputing a new checksum and comparing it with the checksum value stored in the data. Any modification to the bytes protected by a checksum would break the integrity of the original input. Thus, there must be a special branch point (e.g., a conditional jump instruction such as `JZ`, `JE`) in the program, where all well-formed inputs follow the same branch whereas malformed inputs follow the other one. Our goal is to locate such conditional jump instructions. To do this, TaintScope works as follows:

Identifying Potential Checksum Check Points. Since a checksum is usually used to protect a considerable number of contiguous bytes, the recomputed checksum value depends on many input bytes. Consequently, the result of the checksum comparison also depends on many input bytes. For instance, the variable `recomputed_cksum` at line 4 in Figure 3 depends on all bytes in an input image, and thus the result of the comparison at line 6 also depends on the whole input file.

According to this feature, the execution monitor first identifies some potential checksum check points in the target program. The execution monitor instruments all conditional jump instructions in the target program. Before the execution of each conditional jump instruction, the execution monitor checks whether the number of marks associated with the `eflags` register exceeds a predefined threshold value. If so, the conditional jump instruction is considered to be a potential checksum check point. The execution monitor records the relevant information of the instruction in the file *Checksum Info*.

This step may identify many candidates, especially when input data are also compressed or encrypted. A decompressed or decrypted byte usually depends on the whole input. However, programs usually first check the integrity of inputs. We have found empirically that the first candidate is most often the real checksum check point.

Refinement Procedure. In this step, the checksum detector is used to reduce the number of candidate points. Let $\mathcal{A}$ be the set of conditional jump instructions recorded in *Checksum Info*. The checksum detector instruments all instructions in $\mathcal{A}$ to capture their behavior, i.e., whether the conditional jumps are taken or not. These branch profile information can be used to accurately locate checksum check points.

First, the checksum detector runs the program with some well-formed inputs, and then collects and analyzes branch profile data. More specifically, conditional jump instructions that are always taken among all executions are added to set $\mathcal{P}_1$, whereas conditional jump instructions that are always not taken are added to set $\mathcal{P}_0$.

Next, the checksum detector runs the program with some malformed inputs. Note that malformed inputs are generated by modifying well-formed inputs. Similarly, the checksum detector builds another two sets $\mathcal{P}'_1$ and $\mathcal{P}'_0$, where $\mathcal{P}'_1$ and $\mathcal{P}'_0$ contain the always-taken and always-not-taken conditional jump instructions among all executions, respectively. A special case is that an input sample contains multiple checksum fields and these checksums protect different parts of the input sample. For example, a PNG format image consists of many `chunks` and each `chunk` has a CRC checksum. Even if we modify the bytes in the last `chunk`, other `chunks` can still pass the checksum checks. In other words, the real checksum check point may not be included in $\mathcal{P}'_1$ or $\mathcal{P}'_0$. To avoid this problem, the checksum detector can track the propagation of the modified bytes and only count the candidates that are affected by the modified bytes.

Finally, the checksum detector computes the set $(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)$. The conditional jump instructions in $(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)$ behave completely different when the target program runs with well-formed inputs and malformed inputs. The checksum detector outputs such instructions as checksum check points. Furthermore, the checksum detector generates bypass rules in the file *Bypass Info*. A bypass rule mainly consists of an instruction address, an instruction mnemonic and an action (i.e., always-taken or always-not-taken). These bypass rules are used in checksum-aware fuzzing.

Checksum Field Identification. TaintScope can also identify the checksum field in an input instance. Similar to Tupni [34], TaintScope first identifies the trace predicates that can affect the conditional jump instructions in $(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)$, and then locates the predicates of the form `recomputed_chksum==attached_chksum`, where `recomputed_chksum` is a value depending on a considerable number of input bytes and `attached_chksum` is a value only derived from a few input bytes. The input bytes that can affect `attached_chksum` are considered as the checksum field.

D. Directed and Checksum-aware Fuzzing

Directed Fuzzing. The fuzzer module implements the basic functions of a standard fuzzing tool, i.e., it generates malformed inputs, feeds them to the target program, and records the inputs that cause a crash or 100% CPU usage. However, unlike traditional fuzzing tools that blindly change parts of a well-formed input, our fuzzer specially focuses on modifying the hot bytes in a well-formed input. Specially, the fuzzer generates malformed test cases using attack heuristics. For example, hot bytes that can influence memory allocation functions are set to small, large or negative integer values; hot bytes that flow into string functions are replaced by malformed characters, such as `%n`, `%p`.

Directed fuzzing technique can dramatically reduce the size of mutation space because usually only a small portion of input data are hot bytes. Due to the similarity to well-formed inputs, generated test cases satisfy many primitive constraints and have a high probability to test code within the entire program. In addition, since extreme values in generated test cases may directly affect the arguments of system calls or other important APIs, such malformed test cases have a high probability to trigger potential vulnerabilities.

Checksum-aware Fuzzing. If the checksum detector has generated bypass rules, the fuzzer would send malformed inputs to an instrumented program. According to instruction address information in bypass rules, the fuzzer instruments the corresponding branch instructions. To force the conditional jump instructions always (not) taken, the condition code flags (e.g., `OF`, `CF`, `ZF`) in the `eflags` register are set to proper values before the execution of the branch instructions.

However, in practice, considering the overhead of instrumentation, analysts can directly locate the instructions in the raw binary file and modify the binary to ensure the conditional instructions always (not) taken, a classical technique used in software cracking. Then, the fuzzer module feeds malformed inputs to the modified program.

E. Repairing Test Cases Using Combined Concrete and Symbolic Execution

TaintScope needs to fix checksum fields in the malformed test cases that cannot pass checksum checks in the original program but cause the instrumented program to crash.

Since correct checksum fields need to meet the trace constraint $Checksum(D) == T(C_r)$, a direct idea is to collect the constraint by symbolic execution and solve the constraint. However, if all input are replaced by symbolic values, complex checksum algorithms such as MD5 will bring a great challenge to existing solvers to solve the trace constraint [63], [24].

To address this challenge, TaintScope only treats the checksum fields as symbolic values and leaves the majority input bytes as concrete values. Thus, the complex trace constraint $Checksum(D) == T(C_r)$ is simplified to the simple one $c == T(C_r)$, where c is a runtime determinable constant value. The simplified constraint does not depend on checksum algorithms any more. Common transformation functions $T()$, such as conversion from little-endian to big-endian, from hex/oct to decimal numbers, can usually be handled by current constraint solver like STP [39]. Thus, TaintScope can solve the trace constraints and correctly update the checksum fields in malformed test cases.

Specifically, TaintScope first runs the original program with a malformed test case and records the execution trace. When the checksum fields are read into memory, TaintScope specifies these memory addresses in the trace file. Then, TaintScope *offline* symbolically evaluates the recorded trace, like SAGE [43].

When replaying the execution trace, TaintScope maintains a symbolic memory environment, which records a map from concrete addresses/registers to symbolic expressions, and a set of symbolic path conditions, which records the constraints on checksum bytes. The symbolic memory environment would be initialized when checksum fields are read into memory. Subsequently, TaintScope updates the symbolic memory environment and the path condition set according to the semantics of executed X86 instructions.

For the instructions whose all operands are concrete, TaintScope simply updates the execution context. Since only checksum fields are substituted with symbolic values, operands in most instructions are concrete. Moreover, treating checksum fields as symbolic values does not incur symbolic pointer (i.e., a pointer may refer to many objects) problems. That is why our symbolic memory environment only maps concrete memory addresses/registers (instead of symbolic addresses) to symbolic values.

In particular, TaintScope can handle global array reads that contain a symbolic index. TaintScope uses IDAPro [6], a disassembler, to identify global arrays and their size. Assume that IDAPro identifies `0x8000` is the absolute address of a global array `[41,42,43,44]`; for the instruction "`mov eax, [0x8000+ecx*4]`" where `ecx` holds a symbolic value, TaintScope would mark `eax` as a symbolic value and generate a long condition:
`((ecx=0)&&eax=41)||..||((ecx=3)&&eax=44).`

After the execution of the checksum check point (discussed in Section III-C), TaintScope negates the last path condition and solves the path conditions to generate a new test case that can pass checksum checks in the original program. Finally, if the new test case can cause the original program to crash, a potential vulnerability is detected.

F. System Implementation

We have implemented a prototype of TaintScope fuzzing system. TaintScope consists of four modules, the execution monitor, the checksum detector, the fuzzer, and the replayer. The execution monitor, our dynamic taint analysis engine, is built on top of PIN [49]. By intercepting low-level system calls, the execution monitor can identify taint sources. Furthermore, the execution monitor makes use of instruction instrumentation to track the propagation of taint data. By routine instrumentation, the execution monitor is able to check the context (e.g., arguments, registers) of specified APIs before the execution of such APIs. The checksum detector also utilizes PIN's instruction instrumentation to profile the behaviors of branch instructions.

Our fuzzer is a Python program that can mutate well-formed inputs based on hot bytes information and feed malformed test cases to target programs. In particular, the fuzzer leverages the similar attack heuristics used in SPIKE [18], i.e., hot bytes are set to extremal values and malformed strings used by SPIKE.

The replayer, our trace-based symbolic execution module, is responsible to record and replay execution traces, similar to PinSEL [54]. The replayer also implements a script running in IDAPro [6] to export statically-known global array information, and then symbolically re-executes the recorded execution trace. The replayer utilizes STP [39] to build and solve path constraints.

IV. Evaluation

In this section, we present four sets of experiments. In Section IV-B, we evaluated the efficiency of the execution monitor. We sent well-formed images in several formats (e.g., PNG, JPEG, TIFF, BMP, and GIF) to three popular applications (e.g., `ImageMagick`, `Google Picasa`, and `Adobe Acrobat`) and measured the portion of hot bytes

Category	Application	Version	OS	Category	Application	Version	OS
Image Viewer	Google Picasa	3.1.0	Windows	Media Player	MPlayer	SVN-28979	Linux
	Adobe Acrobat	9.1.3	Windows		Gstreamer	0.10.15	Linux
	ImageMagick	6.5.2-7	Linux		Winamp	5.552	Windows
	Microsoft Paint	5.1	Windows	Other	libtiff	3.8.2	Linux
Web Browser	Amaya	11.1	Windows		XEmacs	21.4.22	Linux
	Dillo	2.1.1	Linux		wxWidgets	2.8.10	Linux

Table I
AN INCOMPLETE LIST OF APPLICATIONS USED IN OUR EXPERIMENT

Executable	Package	Input Format	Input Size (Bytes)	# Hot Bytes	# X86 Instrs	Run Time
Display	ImageMagick	TIFF	5778	18	191,759,211	2m53s
			2,020	18	82,640,260	1m30s
		PNG	5,149	9	19,051,746	1m54s
			1,250	29	47,246,043	1m8s
		JPEG	6,617	11	48,983,897	1m13s
			6,934	9	48,823,905	1m11s
PicasaPhotoViewer.exe	Google Picasa	GIF	3,190	14	304,993,501	1m25s
			6,529	43	536,938,567	2m57s
		PNG	2,730	18	712,021,776	5m16s
			1,362	16	660,183,239	4m8s
		BMP	3,174	8	310,909,256	1m21s
			7,462	19	468,273,580	2m35s
Acrobat.exe	Adobe Acrobat	BMP	1,440	6	658,370,048	4m25s
			3,678	6	663,923,080	5m2s
		PNG	770	21	297,492,758	3m8s
			1,250	12	354,685,431	4m31s
		JPEG	1,012	13	328,365,912	4m14s
			2,356	4	356,136,453	4m36s

Table II
HOT BYTES IDENTIFICATION RESULTS

in well-formed inputs. In Section IV-C, we evaluated the effectiveness of our checksum detector. We tested eight applications, which employ different checksum algorithms, such as CRC32, MD5, and Alder32. We applied TaintScope to locate checksum check points in these programs. In Section IV-C, we evaluated the accuracy of checksum field identification and the capability of repairing checksum fields for given test cases. In Section IV-E, we show the vulnerabilities we detected in several widely used applications.

A. Experiment Setup

We apply TaintScope to a large number of real-world applications. An incomplete list of the applications is summarized in Table I. The "OS" column in Table I indicates the operating systems that the applications run on. The applications include popular image viewers (e.g., Google Picasa, ImageMagick), multimedia players (e.g., MPlayer, Winamp), web browsers (e.g., Amaya), widely used libraries (e.g., Libtiff), text editors (e.g., XEmacs), etc.

For applications on Windows platform, our experiments are conducted on a machine with Intel Core 2 Duo CPU at 3.0 GHz and 3.25GB memory, running Windows XP Professional SP3; for applications on Linux platform, our experiments are conducted on a machine with Intel Core 2 Duo CPU at 2.4 GHz and 4.0GB memory, running Fedora Core 10.

B. Hot Bytes Identification

In the first set of experiments, we evaluated the performance of the execution monitor and measured the portion of hot bytes in well-formed inputs. Specially, according to our previous research [65], we found that many integer overflow vulnerabilities are closely related to memory allocation functions. In this experiment, the execution monitor was configured to check the size arguments of memory allocation functions (e.g., malloc, realloc). In other words, input bytes that can flow into memory allocation functions are considered as hot bytes.

Table II shows the results of the evaluation on three widely used applications: ImageMagick, Google Picasa, and Adobe Acrobat. Particularly, ImageMagick and Google Picasa are two popular image viewers available on many platforms. In our experiments, we tested ImageMagick on Linux and Google Picasa on Windows. We also chose Adobe Acrobat because it can convert images to PDF files.

We input well-formed images (including PNG, TIF, JPEG, BMP, and GIF formats) to the three applications and applied the execution monitor to track the propagation of input data. Note that well-formed images were obtained either from the Internet (using Google Image Search) or our local disks.

The "Input Format" and "Input Size" columns in Table II represent the format and size of well-formed images, respectively. We counted the number of hot bytes in well-formed input data, as shown in the "Hot Bytes" column. The size of well-formed inputs is roughly in the range from 1,000 to 7,000 bytes, but the number of hot bytes is less than 50. The reason is that memory allocations during displaying an image usually depend on only a few fields in the image, such as the width, length and color depth fields. We also measured the trace length and performance overhead, as shown in the two rightmost columns. The performance overhead is acceptable. In most cases, instrumented programs took several minutes to display a well-formed image.

C. Checksum Check Points Identification

Another key feature of TaintScope system is that the checksum detector module can automatically locate potential checksum check points in programs. In the second set of experiments, we evaluated the effectiveness of our checksum detector.

File formats and checksum algorithms. We chose six known file formats, as shown in the "File Format" column in Table III. These six file formats employ different checksum algorithms to calculate checksum values. Specifically,

- **PNG**, a popular image format with lossless compression, supports two main types of integrity-checking. First, PNG images are divided into logical data `chunks`, and each `chunk` has an associated CRC stored with it. The integrity of an image can easily be tested without decoding the image. Second, compressed data streams within PNG are stored in the zlib format [35]. Zlib format stores an Adler-32 checksum value of uncompressed data.
- **PCAP** [14], a widely used format for dumping network packet traces, is supported by many packet analyzers, such as Tcpdump, Snort, and Wireshark. Although a PCAP file itself does not contain checksum fields, when parsing a PCAP file, packet analyzers need to check the checksums in TCP/UDP packets in the PCAP file.
- **CVD** [3] is an acronym for ClamAV Virus Database. A CVD file has a 512-bytes header structure, which stores an MD5 checksum value of the whole CVD file. When loading a CVD file, ClamAV first checks the integrity of the CVD file.
- **VCDIFF** (Generic Differencing and Compression Data Format) [46] is a general, efficient and portable data format suitable for delta encoding. `Open-vcdiff` project [13] extends the standard VCDIFF format and Adler32 checksum is used to detect accidental corruption of data.
- **Tar** archive format is widely used on Unix-like systems. A tar archive file is the concatenation of one or more files. Each file in a "tar" package is preceded by a 512-byte header record that contains a checksum value for the whole header.
- **Intel HEX** [7] is a text format, with each line containing hexadecimal values encoding a sequence of data and a checksum value of the data.

Evaluation Methodology. We tested eight applications, as shown in the first and second columns in Table III. These eight applications can process file formats mentioned above. We input well-formed and malformed inputs to target applications, respectively, and applied the execution monitor and checksum detector to locate potential checksum check points in target applications. Specifically,

- We input PNG images to two "closed source" image manipulation applications, `Google Picasa` and `Adobe Acrobat`, and applied TaintScope to locate potential CRC check points in the two applications.
- We input PCAP files to `Tcpdump` and `Snort`. The well-formed PCAP file used in our experiment was obtained by capturing network traffic from our local machine. We specified "`-v tcp -r`" options, which enable `Tcpdump` and `Snort` to read TCP packets from a saved PCAP file and perform packet integrity checks such as verifying the IP header and TCP checksums.
- With "`--info`" option, `sigtool` in `ClamAV` package can verify the integrity of a given CVD file and display detailed information about the file. The well-formed CVD file used in our experiment is `daily.cvd` in `ClamAV` package.
- For VCDIFF and Tar Archive formats, we tested two applications `Open-vcdiff` and GNU `Tar`, respectively. Well-formed inputs used in the two tests were created by the two applications themselves. For example, we first used GNU `Tar` to create a tar archive; then, we applied TaintScope to locate checksum checks when GNU `Tar` extracted files from the tar archive.
- For Intel HEX format, we tested GNU `objcopy`, which can copy the contents of an object file to another. In particular, GNU `objcopy` can translate an object file into another object file in a different format. We applied TaintScope to locate checksum checks when GNU `objcopy` translated an Intel HEX object file into other formats.

Experimental Results. Our checksum identification results are summarized in Table III. TaintScope needs two steps to infer whether/where a binary program performs the checksum checks.

In the first step, TaintScope ran the target program with well-formed test cases and the execution monitor was used to

Executable	Package (Version)	File Format	Checksum Algorithm	$\|\mathcal{A}\|$	$\|(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)\|$	Detected?
PicasaPhotoViewer	Google Picasa (3.1)	PNG	CRC32	830	1	√
Acrobat	Adobe Acrobat (9.1.3)			5,805	1	√
Snort	snort (2.8.4.1)	PCAP	TCP/IP checksum	2	2	√
tcpdump	tcpdump (4.0.0)			5	2	√
sigtool	clamav (0.95.2)	CVD	MD5	2	1	√
vcdiff	open-vcdiff (0.6)	VCDIFF	Adler32	1	1	√
Tar	GNU Tar (1.22)	Tar Archive	Tar checksum	9	1	√
objcopy	GNU binutils (2.17)	Intel HEX	Intel HEX checksum	62	1	√

Table III
CHECKSUM IDENTIFICATION RESULTS

identify potential checksum checks. The column $|\mathcal{A}|$ presents the number of potential checksum checks identified by the execution monitor. Particularly, if a conditional jump instruction depends on more than 16 input bytes, it is considered as a potential checksum check, where the threshold 16 is chosen empirically. For PNG format, the execution monitor identifies a large number of candidates. The main reason is that image data in PNG format are compressed. A byte in decompressed data stream may depend on the entire compressed data. All checks on decompressed data were counted in set $\mathcal{A}$.

In the second step, TaintScope ran the target program with malformed test cases and employed branch profiling techniques to refine the set $\mathcal{A}$. Considering that programs usually check the integrity of inputs first before further processing them, we modified the bytes in well-formed test cases which can affect the first few candidates in the set $\mathcal{A}$ to generate malformed inputs.

The column $|(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)|$ presents the number of final candidates. Manual inspection revealed these candidates were real checksum check points. For `Adobe Acrobat`, TaintScope accurately located the CRC check in a binary file named `ImageConversion.api`. Due to the integrity check failure, `Adobe Acrobat` refused to decompress image data in malformed PNG images and exited immediately. Thus, TaintScope did not locate the Alder32 checksum points. Considering that both `Adobe Acrobat` and `Google Picasa` are closed source applications, we cannot provide more detailed results.

Because we specified "`-v tcp`" options, when parsing a PCAP file, `Tcpdump` and `Snort` first verified the IP header checksum; for TCP packets, `Tcpdump` and `Snort` further verified TCP checksums. While an IP checksum was calculated only for IP header bytes, a TCP checksum was calculated based on the TCP packet. TaintScope accurately located the two checksum check points in `Tcpdump` and `Snort`. Similarly, TaintScope accurately identified the checksum checks in other applications.

The time cost of this phase depends on how many test cases we use. According to our experience, several well-formed test cases and slightly more than ten malformed test cases are enough to locate the checksum check points in target programs. Since processing a test case usually needs several minutes (shown in Table II), this phase can typically be done in tens of minutes.

D. Checksum Fields Identification and Repair

We further evaluated the accuracy of checksum field identification and the capability of repairing checksum fields for malformed test cases. We tested four applications and file formats. The results are shown in Table IV.

The third column in Table IV means the number of checksum fields identified by TaintScope in an input instance; the fourth column means the size (in bytes) of each checksum field identified by TaintScope. For example, TaintScope identified four 4-byte checksum fields in a PNG image. We used 010editor [16], a hex editor with binary templates, to parse the PNG image for verification. The output of 010editor reveals that there are four chunks (e.g., `IHDR`, `PLTE`, `IDAT`, and `IEND`) in the PNG image and each chunk has a 4-byte CRC checksum field.

TaintScope identified eight 2-byte checksum fields in the well-formed PCAP file. For verification, we used Wireshark [21] to parse the PCAP file: there are four TCP packets, and each TCP packet has an IP checksum field and a TCP checksum field.

We also manually compared other outputs of TaintScope with published format specializations. Manual inspection revealed that TaintScope accurately identified the number and the size of checksum fields in each well-formed instance.

The fifth column in Table IV indicates that TaintScope can automatically generate valid checksum fields for given test cases. The experiments proceed as follows. First, we deliberately altered the bytes in checksum fields in such well-formed inputs to generate malformed ones.[1] Next, we input these malformed inputs to corresponding applications, and used TaintScope to record and replay the execution traces. TaintScope treated the bytes in checksum fields as symbolic values. After symbolically re-executing the traces,

[1] Note here we modified checksum fields instead of regular data fields. The reason is that we want to show TaintScope can generate correct/valid checksums for which we have ground truth (the original ones).

TaintScope generated new test cases. In our experiments, these new generated test cases were identical with those original well-formed ones, i.e., TaintScope generated correct checksum fields.

The sixth column in Table IV shows the time (in seconds) TaintScope took to replay recorded traces and solve path constraints. In our experiments, fixing a given test case needs a few minutes, a time-consuming phase.

In short, TaintScope can accurately identify checksum fields and automatically generate valid checksum fields for given test cases.

Executable	File Format	# fields	|field|	Repaired?	Time (s)
display	PNG	4	4	√	271.9
tcpdump	PCAP	8	2	√	455.6
tar	Tar Archive	3	8	√	572.8
objcopy	Intel HEX	4	2	√	327.1

Table IV
CHECKSUM FIELDS IDENTIFICATION AND FIX RESULTS

E. Fuzzing Results

As a fuzzing system, TaintScope has already detected 27 severe vulnerabilities in widely used applications and libraries, such as `Microsoft Paint`, `Adobe Acrobat`, `Google Picasa`, `ImageMagick`, and `Libtiff`. Table V summarizes the experimental results. The "#Vulns" column shows the number of vulnerabilities in corresponding applications and the "Checksum-aware" column indicates whether the vulnerabilities are detected by checksum-aware fuzzing or not.

We manually analyzed the causes of most vulnerabilities and identified five vulnerability types, including buffer overflow, integer overflow, double free, null pointer dereference, and infinite loop. According to our vulnerability reports, Secunia [17] and oCERT [12] have confirmed and published security advisories for most of these vulnerabilities and vendors have also released corresponding patches. The "Advisory" column shows the advisory identifier information. *CVE-xxxx*s represent CVE (Common Vulnerabilities and Exposures) identifiers and *SAxxxx*s are security advisories from Secunia.

The rightmost column shows the Secunia's severity rating for the vulnerabilities. "High" is typically used for remotely exploitable vulnerabilities and "Moderate" is typically used for vulnerabilities that require user interaction. Considering that some vulnerabilities are still in the process of being fixed and/or may be easily exploitable, we do not provide detailed information about these vulnerabilities at this time.

`Adobe Acrobat` can create PDF files from images. TaintScope constructed images (in two different formats) which can cause `Adobe Acrobat` to crash or consume 100% CPU. According to our report and the crashing test case, Secunia has confirmed the memory corruption vulnerability and contacted the vendor. The vendor asked Secunia to postpone the publication of the advisory until a fix is available. We have also confirmed the infinite loop in a binary file named `ImageConversion.api` and already sent the PoC image to Adobe PSIRT (Adobe Product Security Incident Response Team). Adobe has published a security bulletin for the two vulnerabilities and released patches [1].

`Microsoft Paint` has been included in all versions of Microsoft Windows. TaintScope discovered an integer overflow in gdiplus.dll which causes an erroneous memory allocation in `Paint` program when `Paint` opened a malformed JPEG image. Successful exploitation may allow execution of arbitrary code. Microsoft has published a security bulletin MS10-005 [9] according to our report.

TaintScope also discovered an integer overflow and an infinite loop in `Google Picasa`. We first sent the infinite loop information to Google Security Team, however, after the initial contact, we have not heard back from the Google for months. Recently, we found the infinite loop issue has been fixed in its new version. The integer overflow occurs in PicasaPhotoViewer.exe when processing JPEG files, which finally causes a heap buffer overflow.

```
509 wxPNGHandler::LoadFile(wxImage *image,
...//ignore the CRC checks
575 for (i = 0; i < height; i++){
577   if ((lines[i]=(unsigned char*)malloc(width*4))
          == NULL){
579     for ( unsigned int n = 0; n < i; n++ )
580       free( lines[n] ); //first time free()
581     goto error;
...
621 error:
...
630  if ( lines )
631  {
632    for ( unsigned int n = 0; n < height; n++ )
633      free( lines[n] ); //second time free()
```

Figure 5. A double free vulnerability in wxPNGHandler::LoadFile() in wxWidgets 2.8.10

Many vulnerabilities in Table V were detected by our checksum-aware fuzzing techniques, i.e., TaintScope identified and bypassed checksum checks in binary programs, performed directed fuzzing on the instrumented program, and fixed the checksum fields in malformed test cases that caused instrumented programs to crash.

As an example, we present the double free vulnerability [22] in `wxWidgets`, a popular open source cross-platform GUI toolkit. The double free vulnerability can be exploited to potentially execute arbitrary code via a specially crafted PNG file. A basic functionality of `wxWidgets` is to load images in a variety of formats. Specifically, the "`wxPNGHandler::LoadFile()`" function in `wxWidgets` is responsible for loading an image in the PNG format. This function first checks the CRC checksum values in a PNG image, and then processes the

Package	Vuln-Type	# Vulns	Checksum-aware?	Advisory	Severity Rating
Microsoft Paint	Memory Corruption	1	N	CVE-2010-0028	Moderate
Google Picasa	Infinite loop	1	N	pending	N/A
	Integer Overflow	1		SA38435	Moderate
Adobe Acrobat	Infinite loop	1	N	CVE-2009-2995	Extremely critical
	Memory Corruption	1	N	CVE-2009-2989	Extremely critical
ImageMagick	Integer Overflow	1	N	CVE-2009-1882	Moderate
CamlImage	Integer Overflow	3	**Y**	CVE-2009-2660	Moderate
LibTIFF	Integer Overflow	2	N	CVE-2009-2347	Moderate
wxWidgets	Buffer Overflow	2	N	CVE-2009-2369	Moderate
	Double Free	1	**Y**		
IrfanView	Integer Overflow	1	N	CVE-2009-2118	High
GStreamer	Integer Overflow	1	**Y**	CVE-2009-1932	Moderate
Dillo	Integer Overflow	1	**Y**	CVE-2009-2294	High
XEmacs	Integer Overflow	3	**Y**	CVE-2009-2688	Moderate
	Null Dereference	1	N	N/A	N/A
MPlayer	Null Dereference	2	N	N/A	N/A
PDFlib-lite	Integer Overflow	1	**Y**	SA35180	Moderate
Amaya	Integer Overflow	2	**Y**	SA34531	High
Winamp	Buffer Overflow	1	N	SA35126	High
Total		27			

Table V
VULNERABILITIES DETECTED BY TAINTSCOPE

image data row by row. Figure 5 shows the source code snippet of the function. Note that `lines[n]` may be freed twice at lines 580 and 633 if the function `malloc` at line 577 returns a NULL pointer.

In our experiments, TaintScope accurately located the CRC checksum check points in `wxWidgets` and the checksum fields in PNG images. Meanwhile, TaintScope identified the hot bytes in well-formed PNG images which can affect the variable “`width`” used in line 577. In fact, the variable “`width`” corresponds to the width field in a PNG image.

At the fuzzing phase, TaintScope modified the hot bytes to some extremal values and altered the execution flows at the checksum check points. Several malformed test cases triggered the double free vulnerability in the function `wxPNGHandler::LoadFile()`.

At the phase of repairing test cases, TaintScope first recorded the execution trace with a malformed PNG image, and re-executed the trace using symbolic execution. By solving the trace constraints, TaintScope successfully generated a valid checksum value for the malformed PNG image. The new PNG image can pass initial checksum checks in `wxWidgets` and trigger the double-free vulnerability.

Since the width field in a PNG image is protected by checksum values, we believe the vulnerability cannot be detected by random modification.

For more details on the published vulnerabilities in Table V, we refer the readers to Secunia [17]. For instance, Secunia has developed exploits and PoC code for the vulnerabilities in `Gstreamer` [5] and `Winamp` [20] (only available for certain types of vendors and governments).

In summary, our experiments show that:

- Only a small portion of input data are hot bytes. Thus, directed fuzzing can dramatically reduce the mutation space.
- TaintScope can accurately locate the checksum check points in programs and the checksum fields in input instances.
- TaintScope can automatically generate valid checksum fields for malformed test cases.
- By checksum-aware directed fuzzing, TaintScope has successfully detected a number of serious real-world vulnerabilities.

V. DISCUSSION

In this section, we discuss the limitations in the current implementation of TaintScope system.

First, TaintScope currently cannot deal with secure integrity check schemes, such as keyed cryptographic hash algorithms or digital signature, which are designed to protect against intentional data alteration. Although TaintScope can locate and bypass such checks at the checksum-aware fuzzing phase, it is impossible for TaintScope system to automatically generate test cases with valid digital signatures. From software testing point of view, some vulnerabilities could be hidden behind such complex application defenses (e.g., digital signatures). We suggest the software developers disable such defense mechanisms at testing phase. We leave the full study of this problem as our future work.

Second, the effectiveness of TaintScope system may be limited when all input data are encrypted. After data decryption, the complex data dependency relationships will heavily influence hot bytes detection and checksum identification. A mitigation strategy is to configure TaintScope to track the decrypted data only. Recent research such as ReFormat [66] and Dispatcher [25] already shows some promising results to locate encryption/decryption routines. Such techniques could be combined with TaintScope to detect the vulnerabilities when the target program processes the decrypted data.

Third, to infer whether/where a program performs checksum checks, TaintScope relies on both well-formed inputs and malformed inputs. The quality of these inputs also affects the results of checksum check point identifications. In general, target programs (or other third-party programs) are able to produce well-formed samples. Hence, TaintScope obtains malformed inputs by modifying such well-formed samples. Considering that programs usually check the integrity of inputs first before further processing them, TaintScope first identifies some potential checksum check points (see Section III-C), and then modifies the bytes in well-formed samples which can affect these potential checksum check points.

Due to the complexity of the x86 instruction set, the current execution monitor in TaintScope system does not instrument all kinds of x86 instructions. Floating-point instructions and some infrequently used x86 instructions such as `movdqa` are not hooked. In addition, the execution monitor also ignores the control flow dependencies. However, previous study [30] reveals that tracking control flow dependencies may make too many noises. Extending TaintScope to track control flow propagation (similar to [37] and DYTAN [29]) and improving data flow propagation are parts of our future work.

VI. Related Work

Traditional Fuzzing. Fuzzing was first proposed by Miller *et al.* [51]. The simple tool in [51] just generated streams of random characters and sent them to target programs, but it was able to crash 25-33% of the utility programs on UNIX system. Since then, a wide range of fuzzing tools have been developed. Sutton *et al.* [64] presented a recent overview of fuzzing techniques and tools. Traditionally, there are two main ways to get malformed inputs: *data generation* and *data mutation* [58]. The former (e.g., Spike [18], Peach [15], SNOOZE [38]) generates malformed inputs based on predefined specifications and the later (e.g., FileFuzz [4]) mutates well-formed inputs. However, the common drawback of traditional fuzzing techniques is that most malformed inputs are prematurely dropped. To improve the effectiveness of fuzzing tools, the following two categories of new techniques are proposed.

Symbolic-execution-based white-box fuzzing. This technique has been widely implemented in numerous tools, such as CUTE [62], DART [42], SAGE [43], SmartFuzz [52], EXE [28], and KLEE [27]. In general, based on code instrumentation or program tracing, these tools replace concrete input data with symbolic values, collect and solve the constraints on execution traces and guide input error detection and generation. When testing applications with highly-structured inputs, such as compilers and interpreters, Godefroid *et al.* [41] and Majumdar *et al.* [50] proposed a variation technique which employs input symbolic grammar specifications. These tools have proven to highly improve the effectiveness of traditional fuzzing tools. They successfully detected serious bugs in GNU coreutils [27], large shipped Windows applications [43], [41], and Linux file systems [28].

With zero knowledge of the checksum algorithm, automatically generating test cases with correct checksum fields is a great challenge. For simple checksum algorithms such as integer addition, existing symbolic execution systems such as Replayer [56] are able to automatically generate correct checksums. However, as shown in [56], the checksum computation significantly increases the complexity of the collected symbolic formula. Furthermore, previous studies [63] [24] have revealed that current symbolic execution engines and constraint solvers cannot accurately generate and solve the constraints that describe the complete process of complex checksum algorithms. Newsome *et al.* [56] also proposed a reasonable scheme to deal with complex checksum algorithms, i.e., *iteratively* constraining some of the input variables to have the concrete values and then simplifying and solving the trace constraints. Instead, TaintScope *directly* leaves all bytes concrete except those in the checksum fields, which significantly reduces the complexity of the trace constraints.

Taint-analysis-based directed fuzzing. The closest work to ours is BuzzFuzz [40]. BuzzFuzz is a directed dynamic taint-based white-box fuzzing tool. To track taint information, BuzzFuzz needs to instrument an application's *source code*. The instrumented application is responsible for identifying input data that can influence the values at system calls. However, modern applications make extensive use of third-party libraries. BuzzFuzz cannot instrument such libraries if source code is unavailable, leading to the loss of taint information. In contrast, TaintScope directly works on binary executables and can monitor the execution of all libraries. In addition, TaintScope can bypass checksum checks in programs.

Flayer [36] is a taint tracing tool with the ability to redirect the execution flows through the modification of conditional jump instructions. Flayer is based upon functionality from Memcheck [55] and marks input data only with 0/1 label. Thus, Flayer cannot accurately track the impact of input data on an application's execution. Execution flow alteration is also used in analyzing malware behavior, e.g., exploring multiple execution paths [53], and forcing sampled execution to identify various kernel rootkit behaviors [67].

Corpus Distillation [59] is a feedback-driven fuzzing system that uses a code coverage heuristic to select and mutate input samples. In particular, to overcome checksum problems, Corpus Distillation designed the sub-instruction profiling method, i.e., comparison instructions (such as immediate comparison and `rep cmps`) are broken into bit-sized chunks and the coverage score of these instructions depends on the "depth" of comparison. Based on the sub-instruction profiling, Corpus Distillation is able to generate correct CRC checksums in PNG files without the requirement of constraint solving. However, due to the lack of fine-grained taint tracking and checksum field identification, to pass the checksum checks, Corpus Distillation has to mutate all bits in an input sample until reaches checksum fields, which may heavily limit its efficiency.

Moreover, Corpus Distillation and TaintScope can benefit from each other. While TaintScope's checksum check points locating and bypassing techniques can be used in Corpus Distillation to improve the sub-instruction profiling method, the code-coverage-based input sample selection and mutation technique in Corpus Distillation can also be used in the fuzzing phase of TaintScope system.

Many protocol reverser engineering tools (such as Prospex [31], Tupin [34], AutoFormat [47], Polyglot [26], Discoverer [33], FFE/x86 [44]) can be used to guide fuzzing tests. These tools can extract format specifications for input data by analyzing network traffic [33], monitoring the execution of a program while it processes input data [47], [48], [34], [26], [68], or analyzing binary executables directly [44]. The extracted protocol specifications can be further translated into fuzzing specifications. However, none of these systems explicitly discussed how to bypass checksum checks.

In addition, Kang *et al.* [45] proposed a trace matching algorithm to locate the divergence point between two similar traces. This algorithm could also potentially be applied to locate checksum check points in programs.

Finally, there are a significant amount of vulnerability detection studies based on static analysis, instead of dynamic fuzzing. We refer the readers to [8] for further references. In particular, [65] and [32] are two binary analysis tools which can identify integer overflow vulnerabilities or insecure uses of sensitive C library calls in binary executables. While TaintScope is mainly a dynamic fuzzing tool, it could also benefit from the advances in this line of work. For example, TaintScope uses similar heuristics in [65] to specially monitor the size arguments of memory allocation functions and has discovered many integer overflow bugs.

VII. Conclusions

In this paper, we present TaintScope, a checksum-aware directed fuzzing system. Based on fine-grained dynamic taint tracking and branch profiling, TaintScope can monitor the impact of input data on an application's execution, identify the input data that can affect the context of security-sensitive operations, and locate checksum-based integrity checks in programs. TaintScope can dramatically reduce the mutation space and bypass checksum checks by execution flow alteration. Furthermore, TaintScope can automatically fix the checksum fields in malformed test cases using combined concrete and symbolic execution techniques.

We applied TaintScope to a number of large real-world applications. Experimental results show that TaintScope can accurately locate the checksum checks in programs and dramatically improve the effectiveness of fuzz testing. TaintScope has already identified 27 previously unknown vulnerabilities in several widely used applications, including `Microsoft Paint`, `Adobe Acrobat`, `Google Picasa`, and `ImageMagick`. Most of these vulnerabilities have been confirmed by Secunia and oCERT, and have been assigned CVE identifiers (such as CVE-2009-1882, CVE-2009-2688, CVE-2009-2347, and CVE-2009-2369). Corresponding patches from vendors are released or in progress based on our reports.

Acknowledgment

We are grateful to the anonymous reviewers for their hard work, insightful comments and suggestions. This research was supported in part by the National Development and Reform Commission (NDRC) under Project "A monitoring platform for web safe browsing" and the Research Fund for the Doctoral Program of Higher Education of China under Grant No. 200800011019.

References

[1] Adobe Security Bulletins. *http://www.adobe.com/support/security/bulletins/apsb09-15.html.*

[2] Checksum: From Wikipedia encyclopedia. *http://en.wikipedia.org/wiki/Checksum.*

[3] CVD: ClamAV Virus Database. *http://clamav.net/doc/latest/html/node54.html.*

[4] FileFuzz Tool. *http://labs.idefense.com/software/fuzzing.php.*

[5] GStreamer Good Plug-ins PNG Processing Integer Overflow Vulnerability. *http://secunia.com/advisories/35205/.*

[6] Ida pro. *http://www.hex-rays.com/idapro/.*

[7] Intel HEX: From Wikipedia. *http://en.wikipedia.org/wiki/Intel_HEX.*

[8] List of tools for static code analysis. *http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.*

[9] Microsoft security bulletin ms10-005. *http://www.microsoft.com/technet/security/Bulletin/MS10-005.mspx.*

[10] Month of Browser Bugs . *http://browserfun.blogspot.com.*

[11] Month of Kernel Bugs. *http://projects.info-pull.com/mokb/.*

[12] oCERT: Open Source Computer Emergency Response Team. *http://www.ocert.org/*.

[13] open-vcdiff: An encoder/decoder for the VCDIFF (RFC3284) format. *http://code.google.com/p/open-vcdiff/*.

[14] PCAP: Next Generation Dump File Format. *www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html*.

[15] Peach Fuzzing Platform. *http://peachfuzz.sourceforge.net/*.

[16] The professional text/hex editor with binary templates. *http://www.sweetscape.com/010editor/*.

[17] Secunia Website. *http://secunia.com/*.

[18] Spike Fuzzing Platform. *http://www.immunitysec.com/resourcesfreesoftware.shtml*.

[19] Tar: The gnu version of the tar archiving utility. *http://www.gnu.org/software/tar/*.

[20] Winamp MP4 Processing Buffer Overflow Vulnerabilities. *http://secunia.com/advisories/35126/*.

[21] Wireshark: The World's Most Popular Network Protocol Analyzer. *http://www.wireshark.org/*.

[22] wxWidgets Double Free Vulnerabilities. *http://secunia.com/advisories/35292/*.

[23] T. Boutell. PNG (Portable Network Graphics) Specification Version 1.0. RFC 2083, Internet Engineering Task Force, Mar. 1997.

[24] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May, 2008.

[25] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS'09)*, Chicago, IL, November 2009.

[26] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, Oct. 2007.

[27] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, USA, 2008.

[28] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*, pages 322–335, 2006.

[29] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *The 2007 International Conference on Software Testing and Analysis (ISSTA'07)*, 2007.

[30] J. Clause and A. Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *The 2009 International Conference on Software Testing and Analysis (ISSTA'09)*, 2009.

[31] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, USA, 2009. IEEE Computer Society Press.

[32] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static Detection of Vulnerabilities in x86 Executables. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, December 2006.

[33] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium*, 2007.

[34] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: automatic reverse engineering of input formats. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, New York, NY, USA, 2008. ACM.

[35] P. Deutsch and J. l. Gailly. ZLIB compressed data format specification version 3.3. RFC 1950, Internet Engineering Task Force, May 1996.

[36] W. Drewry and T. Ormandy. Flayer: Exposing Application Internals. In *First Workshop On Offensive Technologies (WOOT)*, 2007.

[37] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, June 2007.

[38] G. Banks and M. Cova and V. Felmetsger and K. Almeroth and R. Kemmerer and G. Vigna. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the Information Security Conference (ISC)*, LNCS, Samos, Greece, August 2006. Springer.

[39] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[40] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, New York, NY, USA, 2009. ACM.

[41] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, 2008.

[42] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[43] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[44] J. L. Junghee, T. Reps, and B. Liblit. Extracting output formats from executables. In *Working Conference on Reverse Engineering*, pages 167–178, 2006.

[45] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *Second Workshop on Virtual Machine Security (VMSec)*, Chicago, IL, November 2009.

[46] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The VCDIFF Generic Differencing and Compression Data Format. RFC 3284, Internet Engineering Task Force, June 2002.

[47] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[48] Z. Lin, X. Zhang, and D. Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2008)*, Anchorage, Alaska, USA, June 2008.

[49] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 190–200, New York, NY, USA, 2005. ACM.

[50] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 553–556, New York, NY, USA, 2007. ACM.

[51] B. P. Miller, L. Fredriksen, and S. Bryan. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[52] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[53] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[54] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. *SIGMETRICS Perform. Eval. Rev.*, 34(1):216–227, 2006.

[55] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[56] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, Oct. 2006.

[57] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, Feb. 2005.

[58] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, 3(2):58–62, 2005.

[59] T. Ormandy. Making Software Dumberer. *http://taviso.decsystem.org/making_software_dumber.pdf.*

[60] J. Röning, M. Lasko, A. Takanen, and R. Kaksonen. Protos - systematic approach to eliminate software vulnerabilities. In *Invited presentation at Microsoft Research*, Seattle, USA, 2002.

[61] R. Sekar. An efficient black-box technique for defeating web application attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2009.

[62] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, 2005.

[63] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[64] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[65] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2009.

[66] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS'09)*, September 2009.

[67] J. Wilhelm and T. cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, pages 219–235, 2007.

[68] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *16th Network & Distributed System Security Symposium*, 2008.